Concurrent online tracking of mobile users

Baruch Awerbuch * David Peleg †

Abstract

This paper deals with the problem of maintaining a distributed directory server, that enables us to keep track of mobile users in a distributed network in the presence of concurrent requests. The paper uses the graph-theoretic concept of *regional matching* for implementing efficient tracking mechanisms. The communication overhead of our tracking mechanism is within a polylogarithmic factor of the lower bound.

1 Introduction

Since the primary function of a communication network is to provide communication facilities between users and processes in the system, one of the key problems such a network faces is the need to be able to locate the whereabout of various entities in it. This problem becomes noticeable especially in large networks, and is handled by tools such as name servers and distributed directories (cf. [LEH85, P89b]).

The location problem manifests itself to its fullest extent when users are allowed to relocate themselves from one network site to another frequently and at will, or when processes and servers may occasionally migrate between processors. In this case, it is necessary to have a dynamic mechanism enabling one to keep track of such users and contact them at their current residence. The purpose of this work is to design efficient tracking mechanisms, based on distributed directory structures, minimizing the communication redundancy involved.

Networks with mobile users are by no means far-fetched. A prime example is that of cellular telephone networks. In fact, one may expect that in the future, all telephone systems will be based on "mobile telephone numbers," i.e., ones that are not bound to any specific physical location. Another possible application is a system one may call "distributed yellow pages," or "distributed match-making" [MV88]. Such a system is necessary in an environment consisting of mobile "servers" and "clients." The system has to provide means for enabling clients in need of some service to locate the whereabouts of the server they are looking for. (Our results are easier to present assuming the servers are distinct. However, they are applicable also to the case when a user is actually looking for one of the closest among a set of identical servers.) The method may also find engineering applications in concurrent programming languages, and related areas.

*Department of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139, USA. E-mail: baruch@theory.lcs.mit.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, DARPA contract N00014-89-J-1988, and a special grant from IBM.

†Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. E-mail: peleg@wisdom.bitnet. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Walter and Elise Haas Career Development Award.

In essence, the tracking mechanism has to support two operations: a "move" operation, causing a user to move to a new destination, and a "find" operation, enabling one to contact a specified user at its current address. However, the tasks of minimizing the communication overhead of the "move" and "find" operations appear to be contradictory to each other. This can be realized by examining the following two extreme strategies (considered also in [MV88]).

The *full-information* strategy requires every vertex in the network to maintain a complete directory containing up-to-date information on the whereabouts of every user. This makes the "find" operations cheap. On the other hand, "move" operations are very expensive, since it is necessary to update the directories of all vertices. Thus this strategy is appropriate only for a near static setting, where users move relatively rarely, but frequently converse with each other.

In contrast, the *no-information* strategy opts not to perform any updates following a "move," thus abolishing altogether the concept of directories and making the "move" operations cheap. However, establishing a connection via a "find" operation becomes very expensive, as it requires a global search over the entire network. Alternatively, trying to to eliminate this search, it is possible to require that whenever a user moves, it leaves a "forwarding" pointer at the old address, pointing to the new address. Unfortunately, this heuristic still does not guarantee any good worst-case bound for the "find" operations.

Our purpose is to design some intermediate "partial-information" strategy, that will perform well for any communication/travel pattern, making the costs of both "move" and "find" operations relatively cheap. This problem was tackled also by [MV88]. However, their approach was to consider only the global worst-case performance. Consequently, the schemes designed there treat all requests alike, and ignore considerations such as locality.

Our goal is to design more refined strategies that take into account the inherent costs of the particular requests at hand. It is clear that in many cases these costs may be lower than implied by the global worst-case analysis. In particular, we would like moves to a near-by location, or searches for near-by users, to cost less. (Indeed, consider the case of a person who moves to a different room in the same hotel. Clearly, it is wasteful to update the telephone directories from coast to coast; notifying the hotel operator should normally suffice.) Thus we are interested in the worst case *overhead* incurred by a particular strategy. This overhead is evaluated by comparing the total cost invested in a sequence of "move" and "find" operations against the inherent cost (namely, the cost incurred by the operations themselves, assuming full information is available for free.) This comparison is done over all sequences of "move" and "find" operations. The strategy proposed in this paper guarantees overheads that are *polylogarithmic* in the size and diameter of the network. Our distributed directory does not assume synchrony and allows full concurrency.

Our strategy is based on a hierarchy of *regional* directories, where each regional directory is based on a decomposition of the network into regions. Intuitively, the purpose of the i'th level regional directory is to enable any searcher to track any user residing within distance 2^i from it. This structure is augmented with an elaborate mechanism of forwarding pointers

The organization of a regional directory is based on the graph-theoretic structure of a *regional matching* [AP90a]. An m-regional matching is a collection of sets of vertices, consisting of a *read* set $Read(v)$ and a *write* set $Write(v)$ for each vertex v, with the property that $Read(v)$ intersects with $Write(w)$ for any pair of vertices v, w within distance m of each other. These structures are used to enable localized updates and searches at the regional directories.

In a more general context, regional matchings provide a tool for constructing cheap locality preserving representations for arbitrary networks. For instance, this structure has recently been used in another application, namely, the construction of a network synchronizer with polylogarithmic time and communication overheads [AP90b].

The construction of regional matchings is based on the concept of sparse graph covers [P89a, AP90a].

Such covers seem to play a fundamental role in the design of several types of locality preserving network representations. Indeed, cover-based network representations have already found several applications in the area of distributed network algorithms [P89b, PU89a, PU89b, AGLP89, AP90c, AP90b, AKP90]. Sparse covers and partitions can be constructed via clustering and decomposition techniques developed in [A85, PS89, P89a, AP90a, AP90d, LS91].

The rest of the paper is organized as follows. The next section contains a precise definition of the model and the problem. In Section 3 we give an overview of the proposed solution. The regional directory servers (and the regional matching structure upon which they are based) are described in Section 4. The main, hierarchical directory server is described in Section 5. The mechanism is described under the assumption that "move" and "find" requests arrive sequentially, and Section 6 describes how to extend the solution to allow concurrent accesses.

2 Preliminaries

2.1 The model

We consider the standard model of a point-to-point communication network. The network is described by a connected undirected graph $G = (V, E)$, $|V| = n$. The vertices of the graph represent the processors of the network and the edges represent bidirectional communication channels between the vertices. A vertex may communicate directly only with its neighbors, and messages to non-neighboring vertices are sent along some path connecting them in the graph. It is assumed that efficient routing facilities are provided by the system.

We assume the existence of a *weight* function $w : E \rightarrow \mathcal{R}$, assigning an arbitrary non-negative weight $w(e)$ to each edge $e \in E$. The weight $w(e)$ represents the length of the edge, or the cost of transmitting a message on it.

Our code uses several commands suitable for a distributed environment. The first is "**transfer-control-to** v" which means that the *center of activity (c.o.a.)* is moved to vertex v. When we mention a variable of the protocol, we refer to the variable at the current location of the c.o.a. When we write "`Local_var` $\leftarrow$ **remote-read** `Remote_var` from vertex v", while the c.o.a. is located at u, we mean the following: go from u to v, read variable `Remote_var`, return to u and write the retrieved value into variable `Local_var` at u. At the end of this operation, the c.o.a. remains at u. Similarly, "`Remote_var` at vertex $v \leftarrow$ **remote-write** `Local_var`" means that the value `Local_var` at u is retrieved, and the c.o.a. carries it from u to v and writes it into `Remote_var` variable at v. The c.o.a. then returns to u.

Let us now define some graph notation. For two vertices u, w in G, let $dist(u, w)$ denote the length of a shortest path in G between those vertices, where the length of a path $(e_1, \ldots, e_s)$ is $\sum_{1 \leq i \leq s} w(e_i)$. Let $D(G)$ denote the (weighted) *diameter* of the network G, namely, the maximal distance between any two vertices in G. Throughout we denote $\delta = \lceil \log D(G) \rceil$.

2.2 Statement of the problem

Next, let us define the problem more formally. Denote by $Addr(\xi)$ the current address of the user ξ. A *directory server* $\mathcal{D}$ is a distributed data structure (the *directory*), combined with access protocols that enable one to keep track of the users' movements and to find them whenever needed. Namely, the access protocols enable their users to perform the following two operations.

`Find`(ξ, v) : invoked at the vertex v, this operation delivers a search message from v to the current location $s = Addr(\xi)$ of the user ξ.

`Move`(ξ, s, t) : invoked at the current location $s = Addr(\xi)$ of the user ξ, this operation moves ξ to a new location t and performs the necessary updates in the directory.

For simplicity, we assume at first that individual activations of the operations `Find` and `Move` do not interleave in time, i.e., are performed in an "atomic" fashion. This enables us to avoid issues of concurrency control, namely, questions regarding the simultaneous execution of multiple `Find` / `Move` operations. The necessary modifications for handling the concurrent case are outlined in Section 6.

Communication complexity is measured as follows. The basic message length is $O(\log n)$ bits. (Longer messages are charged proportionally to their length.) The *communication cost* of transmitting a basic message over an edge e is the weight $w(e)$ of that edge. The communication cost of a *protocol* π, denoted $Cost(\pi)$, is the sum of the communication costs of all message transmissions performed during the execution of the protocol.

The assumption of efficient routing facilities in the system is interpreted in this context as follows. Suppose that processor v has to send a message to processor u. Then the message will be sent along a route as short as possible in the network, and the cost of the routing is $O(dist(u,v))$.

We are interested in measuring the communication complexity of the `Find` and `Move` operations in our directories. More specifically, we study the overheads incurred by our algorithms, compared to the minimal "inherent" costs associated with each `Find` and `Move` operation. Consequently, let us first identify these optimal costs.

Consider a `Find` instruction $F = \texttt{Find}(\xi, v)$. Recall that $Cost(F)$ denotes the actual communication cost of F. Define the *optimal cost* of F as $Opt\_cost(F) = dist(v, Addr(\xi))$.

Consider a `Move` instruction $M = \texttt{Move}(\xi, s, t)$. Its actual cost is denoted $Cost(M)$. Let $Reloc(\xi, s, t)$ denote the relocation cost of the user ξ from s to t. We define the *optimal cost* of the operation M as $Opt\_cost(M) = Reloc(\xi, s, t)$, which is the inherent cost assuming no extra operations, such as directory updates, are taken. This cost depends on the distance between the old and new location, and we assume it satisfies $Reloc(\xi, s, t) \geq dist(s,t)$. (In fact, the relocation of a server is typically much more expensive than just sending a single message between the two locations.)

We would like to define the "amortized overhead" of our operations, compared to their optimal cost. For that purpose we consider mixed sequences of `Move` and `Find` operations. Given such a sequence $\bar{\sigma} = \sigma_1, \ldots, \sigma_\ell$, let $\mathcal{F}(\bar{\sigma})$ denote the subsequence obtained by picking only the `Find` operations from $\bar{\sigma}$, and similarly let $\mathcal{M}(\bar{\sigma})$ denote the subsequence obtained by picking only the `Move` operations from $\bar{\sigma}$ (i.e., $\bar{\sigma}$ consists of some shuffle of these two subsequences).

Define the cost and optimal cost of the subsequence $\mathcal{F}(\bar{\sigma}) = (F_1, \ldots, F_k)$ in the natural way, setting

$$\begin{aligned} Cost(\mathcal{F}(\bar{\sigma})) &= \sum_{i=1}^{k} Cost(F_i). \\ Opt\_cost(\mathcal{F}(\bar{\sigma})) &= \sum_{i=1}^{k} Opt\_cost(F_i) \end{aligned}$$

The *find-stretch* of the directory server with respect to a given sequence of operations $\bar{\sigma}$ is defined as

$$Stretch_{find}(\bar{\sigma}) = \frac{Cost(\mathcal{F}(\bar{\sigma}))}{Opt\_cost(\mathcal{F}(\bar{\sigma}))} .$$

The find-stretch of the directory server, denoted $Stretch_{find}$, is the least upper bound on $Stretch_{find}(\bar{\sigma})$, taken over all finite sequences $\bar{\sigma}$.

For the subsequence $\mathcal{M}(\bar{\sigma})$, define the cost $Cost(\mathcal{M}(\bar{\sigma}))$, the optimal cost $Opt\_cost(\mathcal{M}(\bar{\sigma}))$, and the *move-stretch* factors $Stretch_{move}(\bar{\sigma})$ and $Stretch_{move}$ analogously.

We comment that our definitions ignore the initial set-up costs involved in organizing the directory when the user first enters the system.

Finally, define the memory requirement of a directory as the total amount of memory bits it uses in the processors of the network.

We can now formally state our main result. We construct a *hierarchical directory server*, $\mathcal{D}$, guaranteeing $Stretch_{find} = O(\log^2 n)$ and $Stretch_{move} = O(\delta \cdot \log n + \delta^2 / \log n)$ and requiring a total of $O(N \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot \delta \cdot \log^2 n)$ bits of memory (including both data and bookkeeping information) throughout the network, for handling N users, where $\delta = \lceil \log D(G) \rceil$.

3 Overview of the solution

Our scheme is based on a distributed data structure storing pointers to the locations of each user in various vertices. These pointers are updated as users move in the network. In order to localize the update operations on the pointers, we allow some of these pointers to be inaccurate. Intuitively, pointers at locations nearby to the user, whose update by the user is relatively cheap, are required to be more accurate, whereas pointers at distant locations are updated less often.

Our *hierarchical directory server* $\mathcal{D}$ is composed of a hierarchy of $\delta = \lceil \log D(G) \rceil$ *regional directories* $\mathcal{RD}_i$, $1 \leq i \leq \delta$, with regional directories on higher levels of the hierarchy based on coarser decompositions of the network (i.e., decompositions into larger regions). The purpose of the regional directory $\mathcal{RD}_i$ at level i of the hierarchy is to enable a potential searcher to track any user residing within distance 2^i from it.

The regional directory $\mathcal{RD}_i$ is implemented as follows. As in the match-making strategy of [MV88], the mechanism is based on intersecting "read" and "write" sets. A vertex v reports about every user it hosts to all vertices in some specified *write set*, $Write_i(v)$. While looking for a particular user, the searching vertex w queries all the vertices in some specified *read set*, $Read_i(w)$. These sets have the property that the read set of a vertex w is guaranteed to intersect the write set of the vertex v whenever v and w are within distance 2^i of each other. The underlying graph-theoretic structure at the basis of this construction is called a 2^i*-regional matching*. (In contrast, the match-making functions of [MV88] do not have any distance limitation, and they insist on having exactly one element in each intersection.)

Let us now turn to outline the `Move` and `Find` operations of the main, hierarchical directory server. Ideally, whenever the user ξ moves, it should update its address listing in the regional directory $\mathcal{RD}_i$ on all levels $1 \leq i \leq \delta$. Unfortunately, such an update is too costly. To prevent waste in performing a `Move` operation we use a mechanism of "forwarding addresses". Our update policy can be schematically described as follows. Whenever a user ξ moves to a new location at distance d away, only the $\log d$ lowest levels of the hierarchy of regional directories are updated to point directly at the new address. Regional directories of higher levels continue pointing at the old location. In order to help searchers that use these directories (and thus get to the old location), a *forwarding pointer* is left at the old location, directing the search to the new one.

The search procedure thus becomes more involved. Nearby searchers would be able to locate ξ's correct address $Addr(\xi)$ directly, by inspecting the appropriate, low-level regional directory. However, searchers from distant locations that invoke a `Find` operation will fail in locating ξ using the lower-level regional directories (since on that level they belong to a different region). Consequently, they have to use higher levels of the hierarchy. The directories on these levels will indeed have some information on ξ, but this information may be out of date, and lead to some old location $Addr'(\xi)$. Upon reaching $Addr'(\xi)$, the searcher will be redirected to the new location $Addr(\xi)$ through a chain of forwarding pointers. The crucial point is that updates at the low levels are *local*, and thus require low communication complexity.

We shall now proceed with a more detailed treatment of the solution. The example at the end of Section 5 may be of further assistance in clarifying the overall structure of the directory server.

4 Regional directories

4.1 Regional matchings

Our construction revolves on the concept of a *regional matching*. The basic components of this structure are a read set $Read(v) \subseteq V$ and a write set $Write(v) \subseteq V$, defined for every vertex v. Consider the collection $\mathcal{RW}$ of all pairs of sets, namely

$$\mathcal{RW} \; = \; \{ \; Read(v), \; Write(v) \mid v \in V \; \}.$$

The collection $\mathcal{RW}$ is an *m-regional matching* (for some integer $m \geq 1$) if $Write(v) \bigcap Read(u) \neq \emptyset$ for all $v, u \in V$ s.t. $dist(u, v) \leq m$.

For any m-regional matching $\mathcal{RW}$, define the following four parameters:

$Deg_{read}(\mathcal{RW}) = \max_{v \in V} |Read(v)|$,

$Rad_{read}(\mathcal{RW}) = \frac{1}{m} \max_{u,v \in V} \{dist(u,v) \mid u \in Read(v)\}$,

and $Deg_{write}(\mathcal{RW})$, $Rad_{write}(\mathcal{RW})$ are defined analogously based on the sets $Write(v)$.

We rely on the following result.

Theorem 4.1 For all $m, k \geq 1$, it is possible to construct an m-regional matching $\mathcal{RW}_{m,k}$ with

$$\begin{aligned} Deg_{read}(\mathcal{RW}_{m,k}) &\leq 2k \cdot n^{1/k} \\ Deg_{write}(\mathcal{RW}_{m,k}) &= 1 \\ Rad_{read}(\mathcal{RW}_{m,k}) &\leq 2k-1 \\ Rad_{write}(\mathcal{RW}_{m,k}) &\leq 2k-1 \end{aligned}$$

In what follows we use regional matchings in order to design our regional directories. It turns out that the complexities of the `Move` and `Find` operations in these directories depend on the above parameters of the matchings.

4.2 Constructing regional directories

Our directory mechanism is based on hierarchically organizing the tracking information in *regional directories*. A regional directory is based on defining a "regional address" `R_Addr`(ξ) for every user ξ. In the hierarchical context, this address represents the most updated *local* knowledge regarding the whereabouts of the user. In particular, the regional address `R_Addr`(ξ) may be outdated, as ξ may have moved in the meantime to a new location without bothering to update the regional directory.

The basic tasks for which we use the regional directory are similar to those of a regular (global) directory, namely, the retrieval of the regional address, and its change whenever needed. For technical reasons, the modification tasks are easier to represent in the form of "insert" and "delete" operations, rather than the more natural "move" operation. Thus an *m-regional* directory $\mathcal{RD}$ supports the operations `R_find`$(\mathcal{RD}, \xi, v)$, `R_del`$(\mathcal{RD}, \xi, s)$ and `R_ins`$(\mathcal{RD}, \xi, t)$. These operations are defined as follows.

`R_ins`$(\mathcal{RD}, \xi, t)$: invoked at the location t, this operation sets t to be the regional address of ξ, i.e., it sets `R_Addr`$(\xi) \leftarrow t$.

`R_del`$(\mathcal{RD}, \xi, s)$: invoked at the regional address $s =$ `R_Addr`(ξ), this operation nullifies the current regional address of ξ, i.e., sets `R_Addr`$(\xi) \leftarrow nil$.

`R_find`$(\mathcal{RD}, \xi, v)$: invoked at the vertex v, this operation returns (to node v) the regional address `R_Addr`(ξ) of the user ξ. This operation is guaranteed to succeed only if $dist(v, \texttt{R\_Addr}(\xi)) \leq m$. Otherwise, the operation may *fail*, i.e., it may be that no address is found for ξ. If that happens then an appropriate message is returned to v.

The construction of an m-regional directory is based on an m-regional matching $\mathcal{RW}$. The basic idea is the following. Suppose that the regional address of the user ξ is $s =$ `R_Addr`(ξ). Then each vertex u in the write set $Write(s)$ keeps a pointer $\texttt{Pointer}_u(\xi)$, pointing to s.

In order to implement operation `R_find`$(\mathcal{RD}, \xi, v)$, the searcher v successively queries the vertices in its read set, $Read(v)$, until hitting a vertex u that has a pointer $\texttt{Pointer}_u(\xi)$ leading to the regional address of ξ. In case none of the vertices in $Read(v)$ has the desired pointer, the operation is said to end in failure. Note that by definition of an m-regional matching, this might happen only if $dist(v, \texttt{R\_Addr}(\xi)) > m$.

The execution of operation `R_del`$(\mathcal{RD}, \xi, s)$, invoked at $s =$ `R_Addr`(ξ), consists of deleting the pointers $\texttt{Pointer}_u(\xi)$ pointing to s at all the vertices $u \in Write(s)$. Similarly, operation `R_ins`$(\mathcal{RD}, \xi, t)$, invoked at the vertex t, consists of inserting pointers $\texttt{Pointer}_u(\xi)$ pointing to t at all the vertices $u \in Write(t)$, thus effectively setting `R_Addr`$(\xi) = t$. The two operations will be performed together, so ξ cannot end up having more than one address in the regional directory.

A formal presentation of operations `R_find`, `R_del` and `R_ins` is given in Figure 1. The correctness of the above implementation for an m-regional directory can be verified in a straightforward manner from the properties of m-regional matchings. Analysis is deferred to

```
R_find(RD, ξ, v):
    For all u ∈ Read(v) do:
        address ← remote-read Pointer_u(ξ) from vertex u
        If address ≠ nil then Return(address)
    End-for
    If address ≠ nil then Return("failure")


R_del(RD, ξ, s):
    For all u ∈ Write(s) do:
        Pointer_u(ξ) at vertex u ← remote-write nil
    End-for


R_ins(RD, ξ, t)
    For all u ∈ Write(t) do:
        Pointer_u(ξ) at vertex u ← remote-write t
    End-for
```

Figure 1: The three operations of the m-regional directory $\mathcal{RD}$, based on an m-regional matching $\mathcal{RW}$.

the full paper.

5 Hierarchical directory servers

In this section we define our *hierarchical directory server* $\mathcal{D}$, and state its properties and complexity.

5.1 The construction

The hierarchical directory server $\mathcal{D}$ is defined as follows. For every $1 \leq i \leq \delta$, construct a 2^i-regional directory $\mathcal{RD}_i$ based on a 2^i-regional matching as described in the previous subsection. Further, the collection of 2^i-regional matchings used for these regional directories is constructed so that all of them have the same Rad_{read}, Deg_{read}, Rad_{write} and Deg_{write} values, i.e., these parameters are independent of the distance parameter 2^i (the construction described earlier enjoys this independence property).

Each processor v and each user ξ participate in each of the 2^i-regional directories $\mathcal{RD}_i$, for $1 \leq i \leq \delta$. In particular, each vertex v has sets $Write_i(v)$ and $Read_i(v)$ in each $\mathcal{RD}_i$, and each user ξ has a regional address $R\_Addr_i(\xi)$ stored for it in each $\mathcal{RD}_i$. We denote the tuple of regional addresses of the user ξ by

$$\bar{\mathcal{A}}(\xi) = \langle R\_Addr_1(\xi), \ldots, R\_Addr_\delta(\xi) \rangle.$$

As discussed earlier, the regional address $v = R\_Addr_i(\xi)$ (stored at the regional directory of level i) does not necessarily reflect the true location of the user ξ, since ξ may have moved in the meantime to a new location v'. Thus, for every $1 \leq i \leq \delta$ and every user ξ, at any time, the regional address $R\_Addr_i(\xi)$ is either ξ's current address, $Addr(\xi)$, or one of its previous residences. The only variable that is guaranteed to maintain the true current address of ξ is its lowest level regional address, i.e., $R\_Addr_1(\xi) = Addr(\xi)$. (As a rule, the lower the level, the more up-to-date is the regional address.)

This situation implies that finding a regional address of the user ξ alone is not sufficient for locating the user itself. This potential problem is rectified by maintaining at each regional address $R\_Addr_i(\xi)$ a *forwarding pointer* `Forward`(ξ) pointing at some more recent address of ξ. (It should be clear that the user may in the meantime have moved further, and is no longer at the vertex pointed at by `Forward`(ξ).)

The invariant maintained by the hierarchical directory server regarding the relationships between the regional addresses stored at the various levels and the forwarding pointers is expressed by the following definition.

The reachability invariant: The tuple of regional addresses $\bar{\mathcal{A}}(\xi)$ satisfies the *reachability invariant* if for every level $1 \leq i \leq \delta$, at any time, $R\_Addr_i(\xi)$ stores a pointer `Forward`(ξ) pointing to the vertex $R\_Addr_{i-1}(\xi)$.

Thus, the reachability invariant essentially implies that anyone starting at some regional address $R\_Addr_i(\xi)$ and attempting to follow the forwarding pointers will indeed reach the current location of ξ, and moreover, will do so along a path going through all lower-level regional addresses of ξ.

Let us associate with each regional address

$R\_Addr_i(\xi)$ a path denoted by $\texttt{Migrate}_i(\xi)$, which is the actual *migration path* traversed by ξ in its migration from $R\_Addr_i(\xi)$ to its current location, $Addr(\xi)$.

As users move about in the network, the system attempts to maintain its information as accurate as possible, and avoid having chains of long forwarding traces. This is controlled by designing the updating algorithm so that it updates the regional addresses frequently enough so as to guarantee the following invariant.

The proximity invariant: The regional addresses $R\_Addr_i(\xi)$ satisfy the *proximity invariant* if for every level $1 \leq i \leq \delta$, at any time, the distance traveled by ξ since the last time $R\_Addr_i(\xi)$ was updated in $\mathcal{RD}_i$ satisfies

$$|\texttt{Migrate}_i(\xi)| \;\leq\; 2^{i-1} - 1.$$

In order to guarantee the proximity invariant, the vertex $Addr(\xi)$ currently hosting the user ξ maintains also the following two data structures: the tuple of regional addresses $\bar{\mathcal{A}}(\xi)$, and a tuple of *migration counters*

$$\bar{\mathcal{C}}(\xi) \;=\; \langle C_1(\xi), \ldots, C_\delta(\xi)\rangle.$$

Each counter $C_i(\xi)$ counts the distance traveled by ξ since the last time $R\_Addr_i(\xi)$ was updated in $\mathcal{RD}_i$, i.e., $C_i(\xi) = |\texttt{Migrate}_i(\xi)|$. These counters are used in order to decide which regional addresses need to be updated after each move of the user.

5.2 The procedures

Let us next describe the procedures we use. A $\texttt{Find}(\xi, v)$ instruction is performed as follows. The querying vertex v successively issues instructions $\texttt{R\_find}(\mathcal{RD}_i, \xi, v)$ in the regional directories $\mathcal{RD}_1$, $\mathcal{RD}_2$ etc., until it reaches the first level i on which it succeeds. (There must be such a level, since the highest level always succeeds.)

At this point, the searcher v starts tracing the user through the network, starting from $R\_Addr_i(\xi)$, and moving along forwarding pointers. This tracing eventually leads to the real address of the user, $Addr(\xi)$.

$i \leftarrow 0$
$address \leftarrow nil$
Repeat
    $i \leftarrow i + 1$
    $address \leftarrow \texttt{R\_find}(\mathcal{RD}_i, \xi, v)$
Until $address \neq nil$

transfer-control-to vertex $address$
Repeat
    $forward \leftarrow \texttt{Forward}(\xi)$
    transfer-control-to vertex $forward$
Until reaching $Addr(\xi)$

Figure 2: Procedure $\texttt{Find}(\xi, v)$, invoked at the vertex v.

The procedure $\texttt{Find}(\xi, v)$ is formally described in Figure 2.

A $\texttt{Move}(\xi, s, t)$ operation is carried out as follows. All migration counters C_i are increased by $dist(s,t)$. Let C_J be the highest level counter that reaches or exceeds its upper limit $(2^{J-1} - 1)$ as a result of this increase. Then we elect to update the regional directories at levels 1 through J. This involves erasing the old listing of ξ in these directories using procedure $\texttt{R\_del}$ and inserting the appropriate new listing (pointing at t as the new regional address) using $\texttt{R\_ins}$. It is also necessary to leave an appropriate forwarding pointer at $R\_Addr_{J+1}(\xi)$ leading to the new location t, and of course perform the actual relocation of the user (along with its $\bar{\mathcal{A}}(\xi)$ and $\bar{\mathcal{C}}(\xi)$ tuples). The update procedure $\texttt{Move}(\xi, s, t)$ is described in Figure 3.

Detailed correctness proof and analysis are deferred to the full paper. We establish the following.

Lemma 5.1 Given an appropriate family of regional matchings, the hierarchical directory server $\mathcal{D}$ constructed as above satisfies $Stretch_{find} = O(Deg_{read} \cdot Rad_{read})$ and $Stretch_{move} = O(Rad_{write} \cdot Deg_{write} \cdot \delta + \delta^2 / \log n)$, and requires a total of $O(N \cdot Deg_{write} \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot Deg_{read} \cdot \delta \cdot \log n)$ memory bits throughout the network in order to handle N users. ∎

Using Lemma 5.1 and Theorem 4.1, and picking $k = \log n$, we get

Corollary 5.2 The hierarchical directory server $\mathcal{D}$ sat-

For $1 \le i \le \delta$ **do:**
  $C_i(\xi) \leftarrow C_i(\xi) + dist(s,t)$
End-for
$J \leftarrow \max\{i \mid C_i(\xi) \ge 2^{i-1}\}$
`Forward`(ξ) at vertex $R\_Addr_{J+1}(\xi)$ $\leftarrow$ **remote-write** t

For $1 \le i \le J$ **do**:
  $\alpha_i \leftarrow R\_Addr_i(\xi)$
  transfer-control-to vertex α_i
  `Forward`$(\xi) \leftarrow nil$
  `R_del`$(\mathcal{RD}_i, \xi, \alpha_i)$
End-for
Relocate user ξ to vertex t, with $\bar{\mathcal{A}}(\xi)$ and $\bar{\mathcal{C}}(\xi)$ tuples
For $1 \le i \le J$ **do**:
  $R\_Addr_i(\xi) \leftarrow t$
  `R_ins`$(\mathcal{RD}_i, \xi, t)$
  $C_i(\xi) \leftarrow 0$
End-for

Figure 3: Procedure `Move`(ξ, s, t), invoked at $s = Addr(\xi)$.

isfies $Stretch_{find} = O(\log^2 n)$ and $Stretch_{move} = O(\delta \cdot \log n + \delta^2/\log n)$ and uses a total of $O(N \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot \delta \cdot \log^2 n)$ memory bits for handling N users. ∎

Example: Finally let us consider an example case, illustrating the data structures held in the system and the way they are manipulated. The example concerns a searcher v, and a user ξ. This user has initially resided at x_1, then migrated to x_2, then to x_3, and finally to x_4, which is its current address, $Addr(\xi) = x_4$. The initial situation is depicted in Figure 4, including ξ's migration path, the sets $Write_i(\xi)$ and forwarding pointers, and the sets $Read_i(v)$.

Let us first consider a `Find`(ξ, v) request issued by v. Then v will fail to retrieve a pointer for ξ in its queries to the regional directories $\mathcal{RD}_i$ for $i = 1,2,3,4$, since $Read_1(v)$, $Read_2(v)$, $Read_3(v)$ and $Read_4(v)$ do not intersect $Write_1(x_4)$, $Write_2(x_4)$, $Write_3(x_3)$ and $Write_4(x_2)$, respectively. However, it will retrieve the pointer `Pointer`$_z(\xi) = x_2$ stored at the vertex z, since this vertex belongs to $Read_4(v) \cap Write_5(x_2)$. The search will now proceed from x_2 along the forwarding pointers to x_3 and from there to x_4.

Now let us consider move operations. The regional addresses of the user ξ are as illustrated in the figure. Also, the corresponding migration paths are `Migrate`$_1(\xi)$ = `Migrate`$_2(\xi) = \emptyset$, `Migrate`$_3(\xi) = p_1$, `Migrate`$_4(\xi)$ = `Migrate`$_5(\xi) = p_2 \cdot p_1$, and `Migrate`$_6(\xi) = p_3 \cdot p_2 \cdot p_1$, where the lengths of the segments are $|p_1| = 3$, $|p_2| = 2$ and $|p_3| = 20$. Hence the current values of the tuples of regional addresses and migration counters are

$$\bar{\mathcal{A}}(\xi) = \langle x_4, x_4, x_3, x_2, x_2, x_1 \rangle, \quad \bar{\mathcal{C}}(\xi) = \langle 0,0,3,5,5,25 \rangle.$$

(Note that the counters satisfy the proximity invariant.)

Now suppose that the user ξ performs three move operations, as follows: M_1 = `Move`(ξ, x_4, x_5), M_2 = `Move`(ξ, x_5, x_6), and M_3 = `Move`(ξ, x_6, x_7), where $dist(x_4, x_5) = dist(x_5, x_6) = dist(x_6, x_7) = 1$.

Then the data structures of the directory change as follows. In the first move M_1, the counter C_3 is increased to 4, thus exceeding its allowed upper limit. This requires updates to the regional directories $\mathcal{RD}_i$ for $1 \le i \le 3$, which now all point to x_5. More specifically, the pointers `Pointer`$_u(\xi) = x_4$ are erased at all vertices $u \in Write_1(x_4) \cup Write_2(x_4)$, the pointers `Pointer`$_u(\xi) = x_3$ are erased at all vertices $u \in Write_3(x_3)$, and new pointers `Pointer`$_u(\xi) = x_5$ are added at all vertices $u \in Write_1(x_5) \cup Write_2(x_5) \cup Write_3(x_5)$. Also, the forwarding pointer at x_3 is erased (since vertex x_3 ceases to play any role in the directory w.r.t. the user ξ), and the forwarding pointer at x_2 is now directed at x_5. The resulting address and counter tuples of ξ are now

$$\bar{\mathcal{A}}(\xi) = \langle x_5, x_5, x_5, x_2, x_2, x_1 \rangle, \quad \bar{\mathcal{C}}(\xi) = \langle 0,0,0,6,6,26 \rangle.$$

In the second move M_2, the only counter that exceeds its upper limit is C_1. Therefore only $\mathcal{RD}_1$ is updated to lead to x_6, and a new forwarding pointer is added at x_5, directed at x_6. The resulting address and counter tuples of ξ are now

$$\bar{\mathcal{A}}(\xi) = \langle x_6, x_5, x_3, x_2, x_2, x_1 \rangle, \quad \bar{\mathcal{C}}(\xi) = \langle 0,1,1,7,7,27 \rangle.$$

The third move M_3 causes C_4 to overflow. This results in updates to the regional directories $\mathcal{RD}_i$ for $1 \le i \le 4$, which now all point to x_7. The forwarding

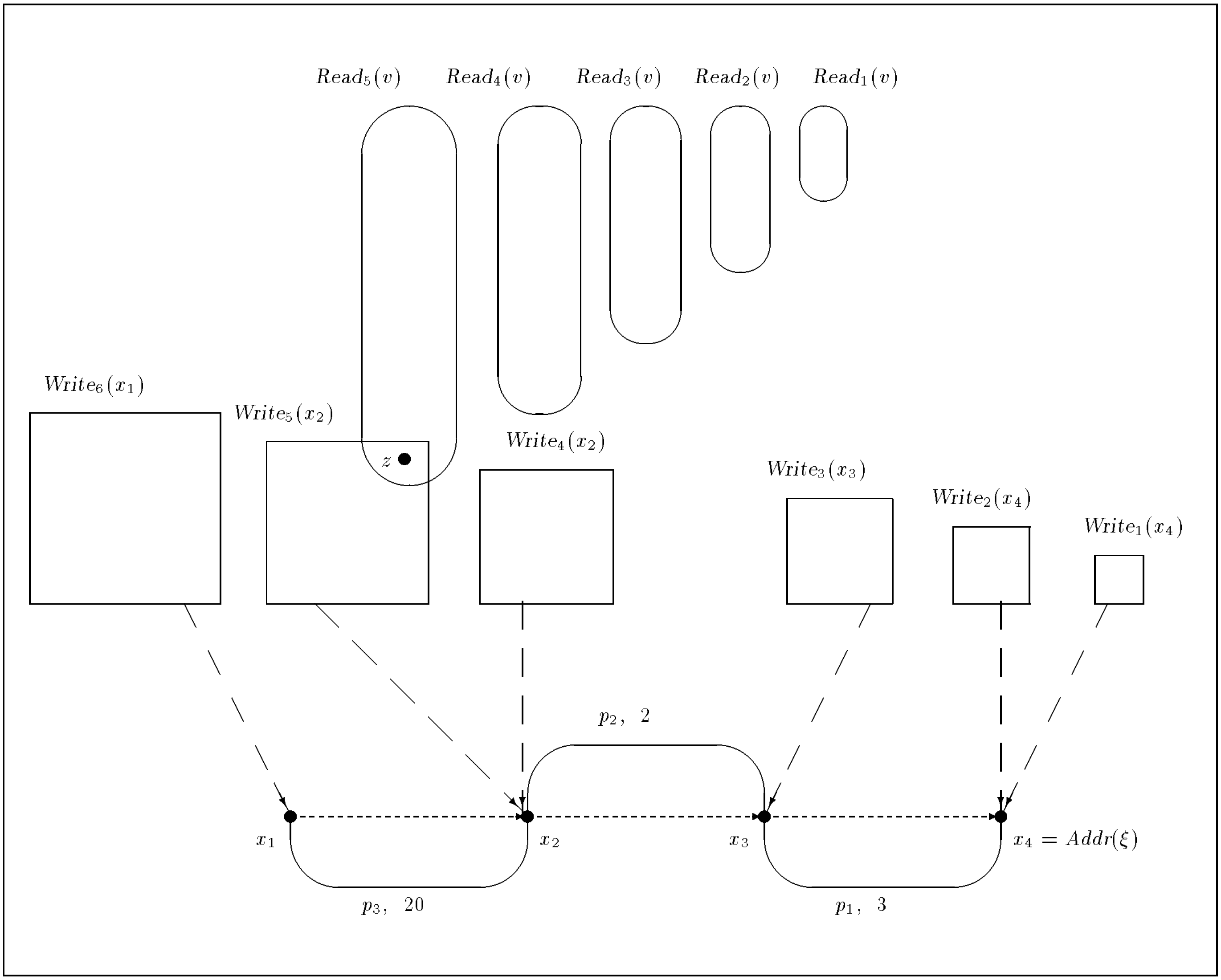

Figure 4: Data structures involved in the example. The solid winding line represents the migration path of the user ξ, composed of three segments p_1, p_2, p_3. The number listed next to each segment represents its length. The dotted arrows represent the forwarding pointers leading to ξ's current residence. The dashed arrows represent the pointers to regional addresses, stored at the appropriate $Write$ sets.

pointer at x_2 is now directed at x_7. The resulting address and counter tuples of ξ are now

$$\bar{\mathcal{A}}(\xi) = \langle x_7, x_7, x_7, x_7, x_2, x_1 \rangle, \ \bar{\mathcal{C}}(\xi) = \langle 0,0,0,0,8,28 \rangle.$$

6 Handling concurrent accesses

Our solution, as described so far, completely ignores concurrency issues. It is based on the assumption that the `Find` and `Move` requests arrive "sequentially," and are handled one at a time, i.e., there is enough time for the system to complete its operation on one request before getting the next one. This assumption would be reasonable if all network communication, as well as all `Move` and `Find` operations, were performed in negligible time. However, in some practical applications, e.g., for satellite links, communication suffers a significant latency. Also, the `Move` and `Find` operations may in fact take a considerable amount of time. In such cases, concurrency issues can no longer be ignored.

Interesting problems arise when many operations are issued simultaneously. Specifically, problems may occur if someone attempts to contact a user *while* it is moving. It is necessary to ensure that the searcher will eventually be able to reach the moving user, even if that user repeatedly moves. In this section we informally outline the particular modifications (both in the model and in the algorithms) needed to handle the case in which operations are performed concurrently and asynchronously.

6.1 Modifications in the model

In order to facilitate reasoning about concurrent operations, we need to introduce some modifications into the model. In particular, it is necessary to address timing issues more explicitly. The input to the system now consists of a stream of (possibly concurrent) requests to perform `Move` and `Find` operations, and the function of the system is to implement these operations. Both `Move` and `Find` operations are viewed as occupying some time interval. A $\texttt{Find}(\xi, v)$ operation starts upon the requesting processor v issuing the request. Its implementation consists of the delivery of a message to the current location of the migrating process, and is terminated at the time of delivery. A $\texttt{Move}(\xi, s, t)$ operation again starts upon the requesting user ξ issuing the request at s. Its implementation consists of the actual move of the process, followed by the updating of various data structures, followed by a signal indicating the termination of the operation. For any operation X, let $T_{start}(X)$ and $T_{end}(X)$ denote the start and termination times of X, respectively. At any given time τ, we define $Addr^{\tau}(\xi)$ to be the current residence of the user ξ at time τ. If at this time ξ is in transit on the way from s to t, then its current residence is considered to be node t.

The concurrent case poses some complications for our cost definitions. In particular, consider a request $F = \texttt{Find}(\xi, v)$. It may so happen that while the directory server attempts to satisfy this request, and deliver the search message from v to ξ, the user ξ itself is busy migrating, in some arbitrary direction. In fact, ξ could possibly perform several moves while searched by v. How then should we define the *inherent* cost of the search? The approach adopted here is the following. First, the correctness requirement of the directory server is that a `Find` operation F always terminates successfully within finite time, i.e., the "chase" cannot proceed forever, and $T_{end}(F) < \infty$.

The operation F takes place in the time interval $[T_{start}(F), T_{end}(F)]$. Since the user ξ may have moved (perhaps more than once) during this period, we redefine the optimal cost of this operation to be the maximal distance from v to any location occupied by ξ throughout the duration of the operation, namely

$$Opt\_cost(F) = \max_{T_{start}(F) \le \tau \le T_{end}(F)} \{dist(v, Addr^{\tau}(\xi))\}.$$

Despite its seeming permissiveness, this definition is in fact quite reasonable, as can be realized by considering some naturally occurring scenarios.

6.2 Algorithmic modifications

Let us next discuss the necessary algorithmic modifications. The problems that arise in the concurrent case can be classified into two types, roughly corresponding to the two parts of procedure $\texttt{Find}(\xi, v)$. The first part involves the retrieval of some regional address $R\_Addr_i(\xi)$ of the user. The second involves proceeding from that address to "trace down" the user. This second part can be thought of as sending a "tracing message" to chase the user, along forwarding pointers.

The idea is that in order to prevent endless chases, the invariant that we would like to preserve is that the searcher is allowed to "miss" the user while searching for it on level i only if the user is currently on transition to a new location farther away than distance 2^i. If the user is currently moving *within* the 2^i vicinity of the searcher, then it must be found.

In order to enforce such invariant, it is necessary to make sure that both parts of the `Find` procedure succeed. First, the searcher should be able to retrieve a regional address of the user at level i. Secondly, once such an address is retrieved, it should suffice to lead the tracing message to the user within a "short" chase (where "short" is to be understood in accordance with our stretch bounds). Specifically, this is imposed by ensuring that following a move that involved updating regional directories up to level ℓ, the user ξ is not allowed to start a new move before it is found by any searcher that is already at the second stage of the search, i.e., that has already retrieved some current regional address $R\_Addr_i(\xi)$, for $i \leq \ell + 1$.

Our algorithms have the same general structure as before, except for some minor modifications. Most of the changes involve permuting and altering some of the steps in the implementation of $\texttt{Move}(\xi, s, t)$. The first two changes handle the second problem, of enabling the tracing message to reach the user, once some regional address has been obtained. To begin with, the user ξ first registers in its new address t, before deleting its registration at the old address s. Thus, ξ may be temporarily "doubly-registered" at both the new and old addresses. As a result, the $\texttt{Pointer}(\xi)$ mechanism is not necessarily a single pointer any longer, but rather a collection possibly containing two pointers.

A second change is that the old regional addresses are deleted top-down, i.e., starting from the highest-level regional directory and ending with the lowest-level one. Along its way, the deletion process also "swaps" the route and verifies that there are no tracing messages for ξ in transit. A similar change is made in procedure $\texttt{R\_del}(\mathcal{RD}_i, \xi, s)$. Implementation of the swap operation is omitted.

A third change required for the concurrent implementation involves the `R_find` operation in a regional directory of level i, and addresses the first part of the `Find` procedure. I.e., the purpose of this change is to guarantee the retrieval of the i'th level regional address $R\_Addr_i(\xi)$ of the user ξ even while this address is being changed, as long as both the new and the old addresses are within distance 2^i of v.

To see where a problem might arise, consider a node v invoking $\texttt{R\_find}(\mathcal{RD}_i, \xi, v)$, while $R\_Addr_i(\xi)$ is changed from s' to t', such that both $dist(v, s') \leq 2^i$ and $dist(v, t') \leq 2^i$. The implementation of this operation consists of the **remote-read** of $\texttt{Pointer}_u(\xi)$ from u, for all $u \in Read_i(v)$. Even though deleting the pointers $\texttt{Pointer}_u(\xi) = s'$ at all $u \in Write_i(s')$ (in sub-operation $\texttt{R\_del}(\mathcal{RD}_i, \xi, s')$) is performed *after* adding the pointers to t', $\texttt{Pointer}_u(\xi) = t'$, at all $u \in Write_i(t')$ (in sub-operation $\texttt{R\_ins}(\mathcal{RD}_i, \xi, t')$), it is still easy to design a scenario in which all of the **remote-read** operations done by v fail to detect a pointer to ξ. This type of "race problems" is typical to asynchronous systems. In our case, there are two ways to go about solving this problem. The first is based on strengthening the definition of m-regional matchings. Specifically, let us introduce the additional requirement that for every v, u_1 and u_2, if $dist(v, u_1) \leq m$ and $dist(v, u_2) \leq m$, then

$$Read(v) \cap Write(u_1) \cap Write(u_2) \neq \emptyset.$$

This would eliminate the difficulty outlined above, since when v queries all nodes in $Read_i(v)$, it hits also some node $x \in Read_i(v) \cap Write_i(s') \cap Write_i(t')$, and at this node, the set $\texttt{Pointer}_x(\xi)$ must contain at least one of s' and t'.

Yet a stronger requirement that can be imposed on an m-regional matching is that for every v and u, if $dist(v, u) \leq m$ then $Read(v) \subseteq Write(u)$. This last requirement clearly implies the former one. In the full paper we show how this property is achieved by slightly modifying the construction of regional matchings. This modification results also in (minor) changes in the stretch complexities of the "find" and "move" operations (in particular, a $\log n$ factor is shifted from $Stretch_{find}$ to $Stretch_{move}$).

Our second, more traditional approach to solving the problem, does not involve any changes in complexity. This approach is based on the observation that the difficulty would have been resolved if **remote-read** operations from v to nodes in $Read(v)$ were performed at the same instance of time. Unfortunately, simultaneous actions are impossible to coordinate in asynchronous systems; however, "causal independence" [L78] is typically sufficient. This type of independence can be achieved in our case. Details are deferred to the full paper.

References

[AGLP89] B. Awerbuch, A. Goldberg, M. Luby and S. Plotkin, Network decomposition and locality in distributed computation, *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989.

[AKP90] B. Awerbuch, S. Kutten and David Peleg. On buffer-economical store-and-forward deadlock prevention, *Proc. INFOCOM*, 1991.

[A85] B. Awerbuch, Complexity of network synchronization, *J. of the ACM* **32**, (1985), 804–823.

[AP90a] B. Awerbuch and D. Peleg, Sparse Partitions, *31st IEEE Symp. on Foundations of Computer Science*, Oct. 1990, 503–513.

[AP90b] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead, *31st IEEE Symp. on Foundations of Computer Science*, 1990, pp. 514–422.

[AP90c] B. Awerbuch and D. Peleg, Routing with polynomial communication-space trade-off, *SIAM J. on Discrete Math.*, to appear.

[AP90d] B. Awerbuch and D. Peleg, *Efficient Distributed Construction of Sparse Covers*, Technical Report CS90-17, The Weizmann Institute, July 1990.

[L78] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Comm. of the ACM* **21**, (1978), 558–565.

[LEH85] K.A. Lantz, J.L. Edighoffer and B.L. Histon, Towards a Universal Directory Service, *4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 261–271.

[LS91] N. Linial and M. Saks, Decomposing Graphs Into Regions of Small Diameter, *2nd ACM Symp. on Discrete Algorithms*, San Francisco, 1991.

[MV88] S.J. Mullender and P.M.B. Vitányi, Distributed Match-Making, *Algorithmica* **3**, (1988), pp. 367–391.

[PS89] D. Peleg and A.A. Schäffer, Graph spanners, *J. of Graph Theory* **13**, (1989), 99–116.

[PU89a] D. Peleg and J.D. Ullman, An optimal synchronizer for the hypercube. *SIAM J. on Comput.* **18**, (1989), 740–747.

[PU89b] D. Peleg and E. Upfal, A tradeoff between size and efficiency for routing tables, *J. of the ACM* **36**, (1989), 510–530.

[P89a] D. Peleg, Sparse Graph Partitions, Report CS89-01, Dept. of Applied Math., The Weizmann Institute, Rehovot, Israel, February 1989.

[P89b] D. Peleg, Distance-Dependent Distributed Directories, *Information and Computation*, to appear. Also as Report CS89-10, Dept. of Applied Math., The Weizmann Institute, Rehovot, Israel, May 1989.